UNIT 4

DYNAMIC PROGRAMMING

Patricia Cuesta Ruiz

Isabel Blanco Martínez

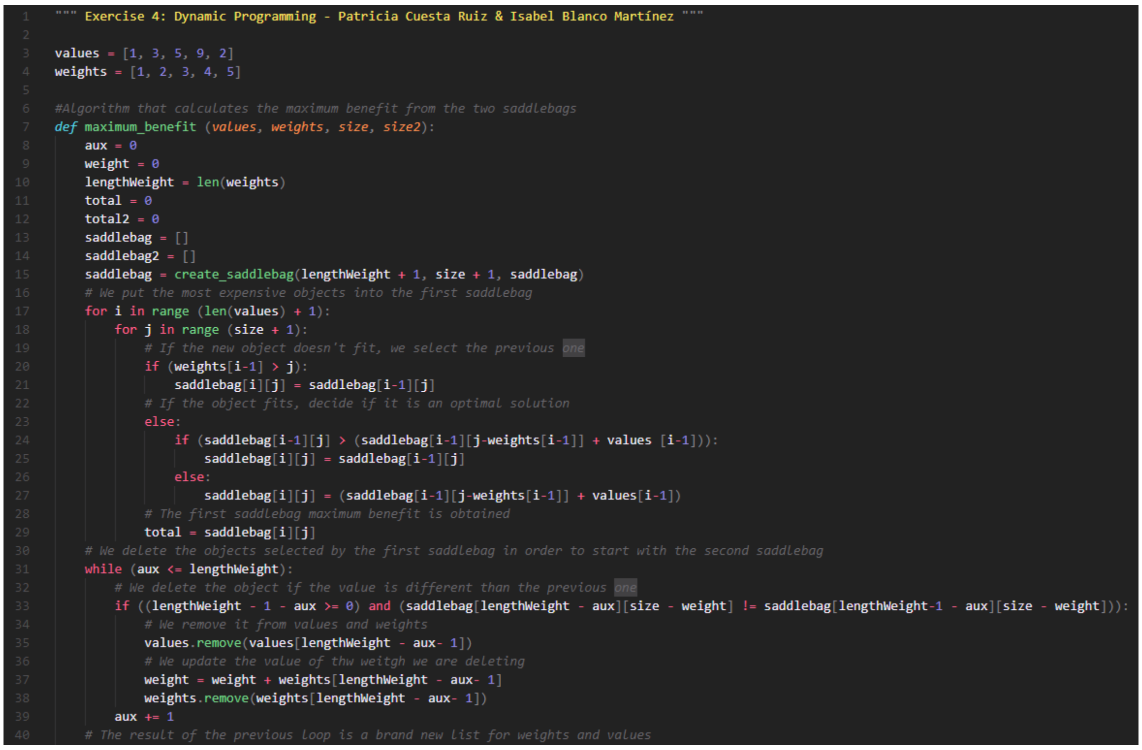
EXERCISE 4

Ali Baba has managed to enter the cave of the one hundred and one thousand thieves, and has brought his camel along with two large panniers. The problem is that he finds so much treasure that he does not know what to take. The treasures are carved jewels, works of art, ceramics ... that is, they are unique objects that cannot be split since then their value would be reduced to zero.

Fortunately, the thieves have everything very well organized and find a list of all the treasures in the cave, which reflects the weight of each piece and its value in the Damascus market. For his part, Ali knows the weight capacity of each of the saddlebags.

Design an algorithm that, taking as data the weights and value of the pieces, and the capacity of the two saddlebags, allows to obtain the maximum benefit that Ali Baba can get from the cave of wonders.

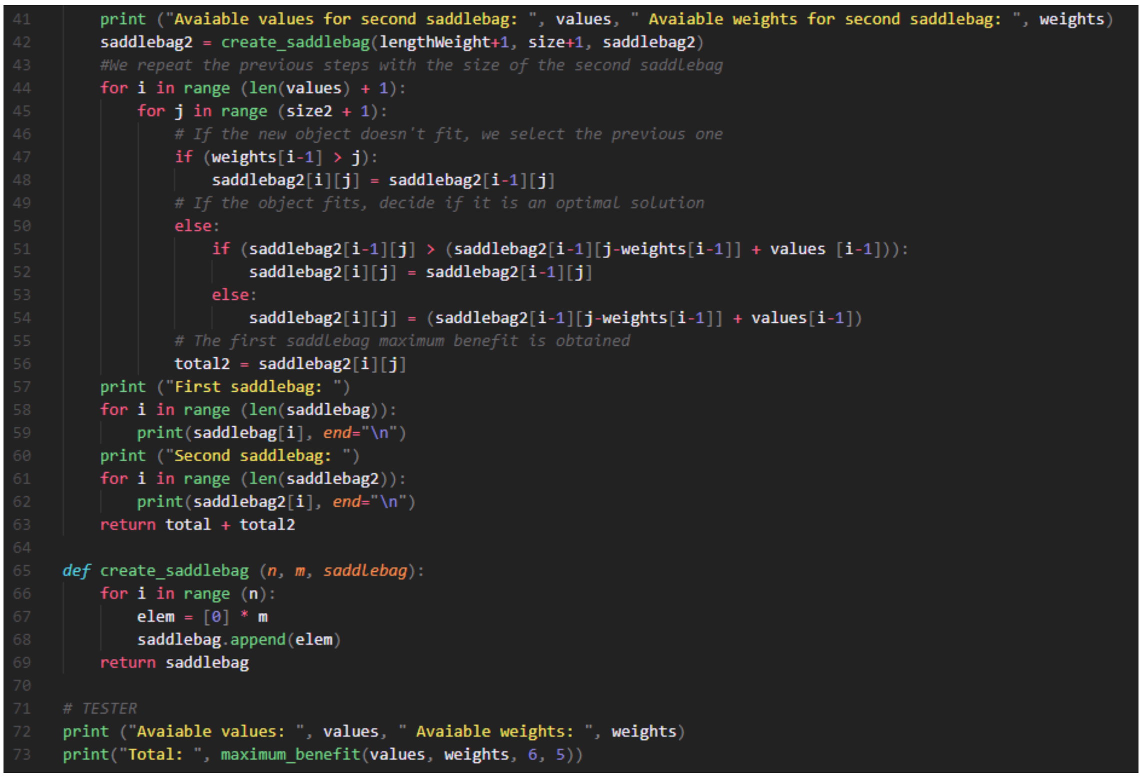
Code:



```python
""" Exercise 4: Dynamic Programming - Patricia Cuesta Ruiz & Isabel Blanco Martínez """

values = [1, 3, 5, 9, 2]
weights = [1, 2, 3, 4, 5]

#Algorithm that calculates the maximum benefit from the two saddlebags
def maximum_benefit (values, weights, size, size2):
    aux = 0
    weight = 0
    lengthWeight = len(weights)
    total = 0
    total2 = 0
    saddlebag = []
    saddlebag2 = []
    saddlebag = create_saddlebag(lengthWeight + 1, size + 1, saddlebag)
    # We put the most expensive objects into the first saddlebag
    for i in range (len(values) + 1):
        for j in range (size + 1):
            # If the new object doesn't fit, we select the previous one
            if (weights[i-1] > j):
                saddlebag[i][j] = saddlebag[i-1][j]
            # If the object fits, decide if it is an optimal solution
            else:
                if (saddlebag[i-1][j] > (saddlebag[i-1][j-weights[i-1]] + values [i-1])):
                    saddlebag[i][j] = saddlebag[i-1][j]
                else:
                    saddlebag[i][j] = (saddlebag[i-1][j-weights[i-1]] + values[i-1])
            # The first saddlebag maximum benefit is obtained
            total = saddlebag[i][j]
    # We delete the objects selected by the first saddlebag in order to start with the second saddlebag
    while (aux <= lengthWeight):
        # We delete the object if the value is different than the previous one
        if ((lengthWeight - 1 - aux >= 0) and (saddlebag[lengthWeight - aux][size - weight] != saddlebag[lengthWeight-1 - aux][size - weight])):
            # We remove it from values and weights
            values.remove(values[lengthWeight - aux- 1])
            # We update the value of thw weitgh we are deleting
            weight = weight + weights[lengthWeight - aux- 1]
            weights.remove(weights[lengthWeight - aux- 1])
        aux += 1
    # The result of the previous loop is a brand new list for weights and values
```

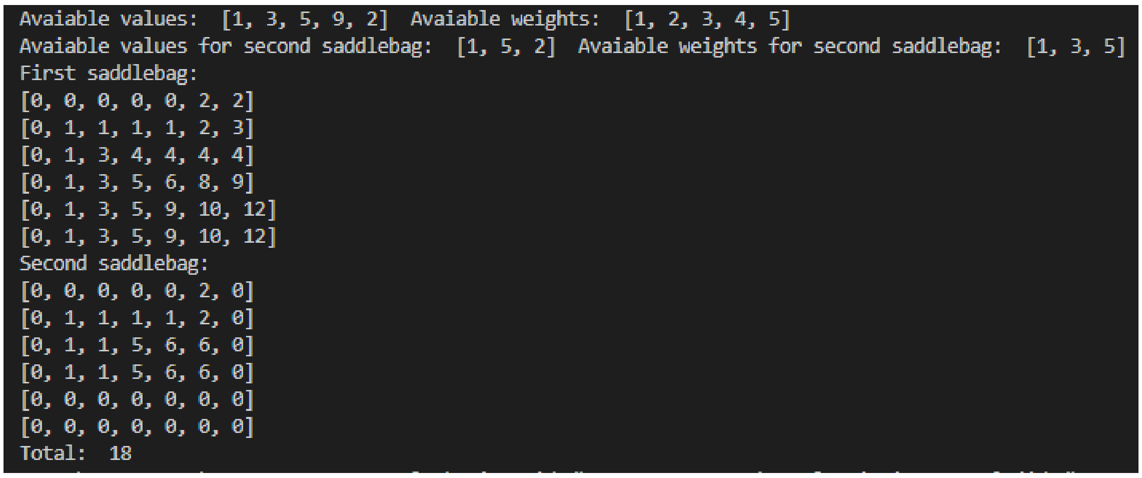


```
    print ("Avaiable values for second saddlebag: ", values, " Avaiable weights for second saddlebag: ", weights)
    saddlebag2 = create_saddlebag(lengthWeight+1, size+1, saddlebag2)
    #We repeat the previous steps with the size of the second saddlebag
    for i in range (len(values) + 1):
        for j in range (size2 + 1):
            # If the new object doesn't fit, we select the previous one
            if (weights[i-1] > j):
                saddlebag2[i][j] = saddlebag2[i-1][j]
            # If the object fits, decide if it is an optimal solution
            else:
                if (saddlebag2[i-1][j] > (saddlebag2[i-1][j-weights[i-1]] + values [i-1])):
                    saddlebag2[i][j] = saddlebag2[i-1][j]
                else:
                    saddlebag2[i][j] = (saddlebag2[i-1][j-weights[i-1]] + values[i-1])
            # The first saddlebag maximum benefit is obtained
            total2 = saddlebag2[i][j]
    print ("First saddlebag: ")
    for i in range (len(saddlebag)):
        print(saddlebag[i], end="\n")
    print ("Second saddlebag: ")
    for i in range (len(saddlebag2)):
        print(saddlebag2[i], end="\n")
    return total + total2

def create_saddlebag (n, m, saddlebag):
    for i in range (n):
        elem = [0] * m
        saddlebag.append(elem)
    return saddlebag

# TESTER
print ("Avaiable values: ", values, " Avaiable weights: ", weights)
print("Total: ", maximum_benefit(values, weights, 6, 5))
```

Output:



```
Avaiable values:  [1, 3, 5, 9, 2]  Avaiable weights:  [1, 2, 3, 4, 5]
Avaiable values for second saddlebag:  [1, 5, 2]  Avaiable weights for second saddlebag:  [1, 3, 5]
First saddlebag:
[0, 0, 0, 0, 0, 2, 2]
[0, 1, 1, 1, 1, 2, 3]
[0, 1, 3, 4, 4, 4, 4]
[0, 1, 3, 5, 6, 8, 9]
[0, 1, 3, 5, 9, 10, 12]
[0, 1, 3, 5, 9, 10, 12]
Second saddlebag:
[0, 0, 0, 0, 0, 2, 0]
[0, 1, 1, 1, 1, 2, 0]
[0, 1, 1, 5, 6, 6, 0]
[0, 1, 1, 5, 6, 6, 0]
[0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
Total:  18
```

Explication:

This exercise is very similar to the backpack example shown in class. The main difference is that this exercise uses two saddlebags instead of just one.

It is important to consider that having two saddlebags means that the objects kept in the first saddlebag will no longer be available for the second saddlebag, so those objects must me removed.

We have taken this into account, so that objects that have been used are identified in order to remove them easily before filling the second saddlebag. The procedure to fill the second saddlebag is the same followed with the first one but using different objects.

We have used arrays and loops that allows us to obtain the objects that the saddlebag must contain from the bag's table.

After that we take the maximum value for each weight and return the sum of the two.

All the steps followed are explained in the code.

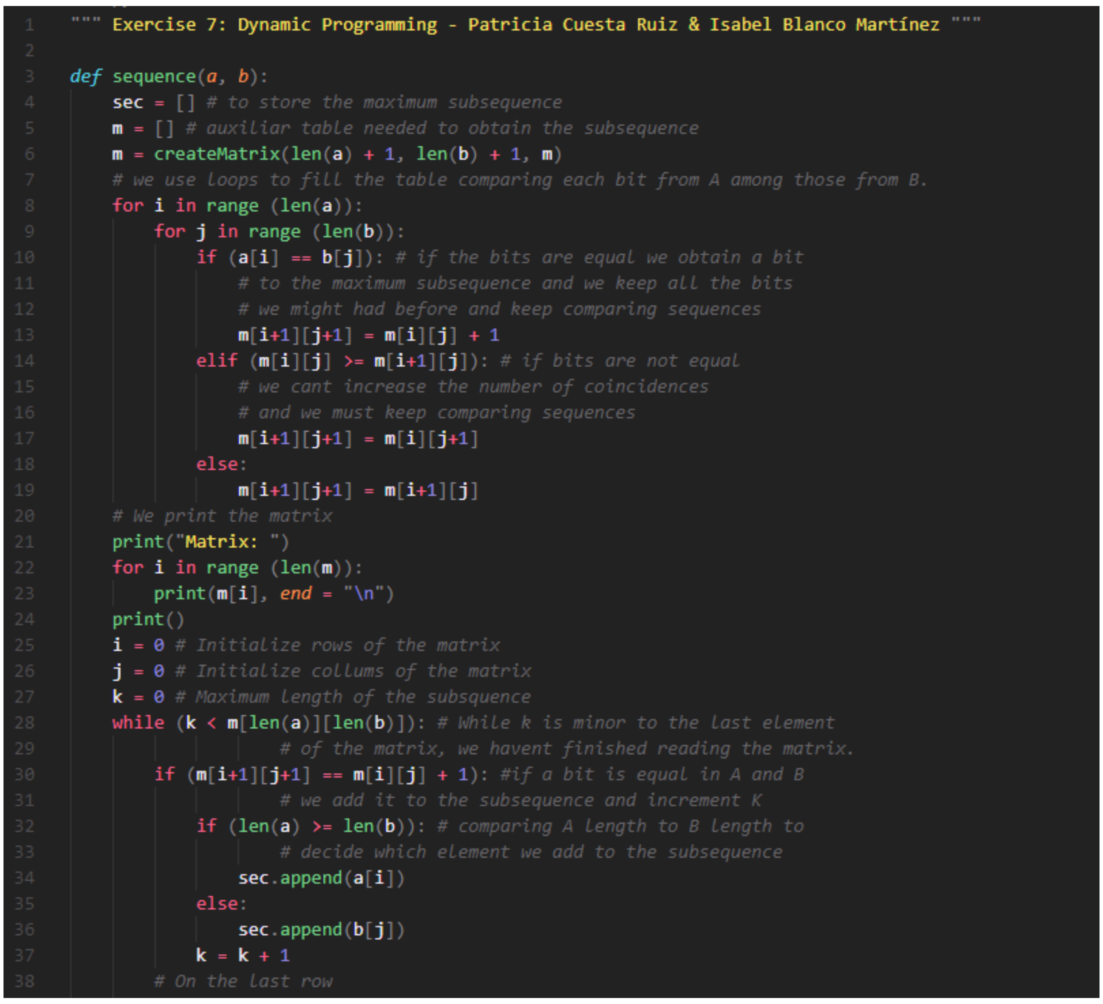
EXERCISE 7:

A sequence of bits A is defined as a sequence A = {a_1, a_2,..., a_n} where each a_i can take the value 0 or the value 1, and n is the length of the sequence A. From a sequence it is defined a subsequence X of A as X = {x_1, x_2,..., x_k}, where k<=n, so that X can be obtained by eliminating some element of A but respecting the order in which the bits appear; for example, if A = {1,0,1,1,0,0,1} we could obtain as subsequences {1,1,1,0,1}, {1,0,1} or 1,1,0,0} among others, but you could never get the subsequence {1,0,0,1,1}.

Given two sequences A and B, X is called a common subsequence of A and B when X is a subsequence of A and is also a subsequence of B. (although they may have been obtained by removing different elements in A than B, and even different quantities of elements). Assuming the sequences A = {0,1,1,0,1,0,1,0} and B = {1,0,1,0,0,1,0,0,1}, a common subsequence would be X = {1,1,0,1}, but it could not be X = {0,1,1,1,0}. We want to determine the common subsequence of two sequences A and B that have the maximum length, for which it is requested:

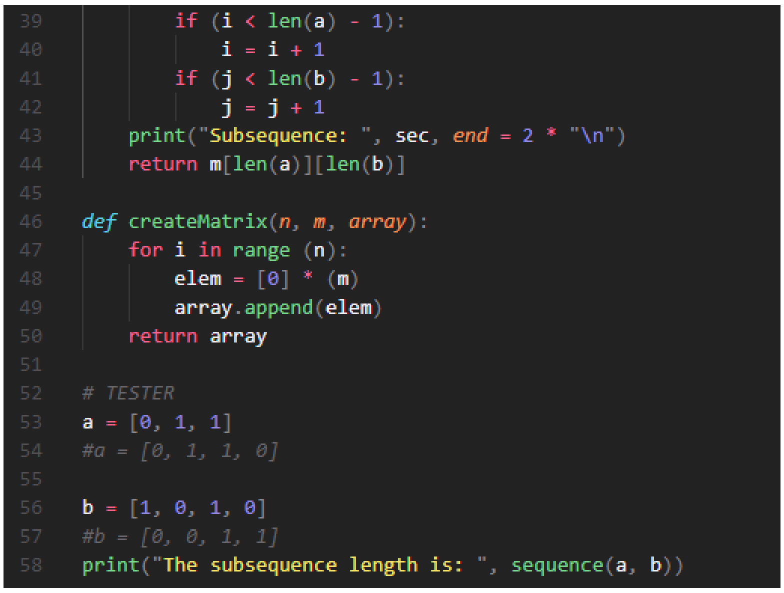
- explain in detail how to solve the problem, and
- make a Dynamic Programming algorithm that obtains the maximum possible length and a common sequence of that length.

Code:



```python
""" Exercise 7: Dynamic Programming - Patricia Cuesta Ruiz & Isabel Blanco Martínez """

def sequence(a, b):
    sec = [] # to store the maximum subsequence
    m = [] # auxiliar table needed to obtain the subsequence
    m = createMatrix(len(a) + 1, len(b) + 1, m)
    # we use loops to fill the table comparing each bit from A among those from B.
    for i in range (len(a)):
        for j in range (len(b)):
            if (a[i] == b[j]): # if the bits are equal we obtain a bit
                # to the maximum subsequence and we keep all the bits
                # we might had before and keep comparing sequences
                m[i+1][j+1] = m[i][j] + 1
            elif (m[i][j] >= m[i+1][j]): # if bits are not equal
                # we cant increase the number of coincidences
                # and we must keep comparing sequences
                m[i+1][j+1] = m[i][j+1]
            else:
                m[i+1][j+1] = m[i+1][j]
    # We print the matrix
    print("Matrix: ")
    for i in range (len(m)):
        print(m[i], end = "\n")
    print()
    i = 0 # Initialize rows of the matrix
    j = 0 # Initialize collums of the matrix
    k = 0 # Maximum length of the subsequence
    while (k < m[len(a)][len(b)]): # While k is minor to the last element
                # of the matrix, we havent finished reading the matrix.
        if (m[i+1][j+1] == m[i][j] + 1): #if a bit is equal in A and B
                # we add it to the subsequence and increment K
            if (len(a) >= len(b)): # comparing A length to B length to
                # decide which element we add to the subsequence
                sec.append(a[i])
            else:
                sec.append(b[j])
            k = k + 1
        # On the last row
```



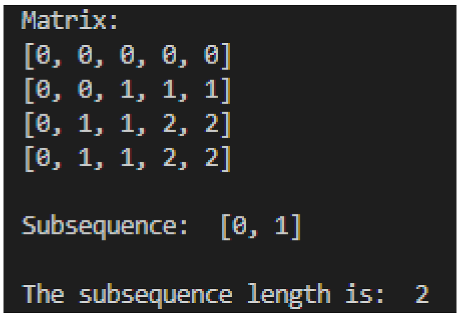
```python
        if (i < len(a) - 1):
            i = i + 1
        if (j < len(b) - 1):
            j = j + 1
    print("Subsequence: ", sec, end = 2 * "\n")
    return m[len(a)][len(b)]

def createMatrix(n, m, array):
    for i in range (n):
        elem = [0] * (m)
        array.append(elem)
    return array

# TESTER
a = [0, 1, 1]
#a = [0, 1, 1, 0]

b = [1, 0, 1, 0]
#b = [0, 0, 1, 1]
print("The subsequence length is: ", sequence(a, b))
```

Output:



```
Matrix:
[0, 0, 0, 0, 0]
[0, 0, 1, 1, 1]
[0, 1, 1, 2, 2]
[0, 1, 1, 2, 2]

Subsequence:  [0, 1]

The subsequence length is:  2
```

Explanation:

To solve this exercise, we receive two sequences A and B, and we have to obtain the subsequence with a length K.

In order to do that, we are using a matrix M[i][j] as a table that we use to obtain the subsequence. Each M[i][j] shows us the length of the maximum subsequence found among the i first bits of the sequence A and the j first elements of the sequence B.

If the bits A[i] and B[j] are the same, we obtain a common bigger bit to the subsequence and determine its length.

We read the matrix using a while until the subsequence is finished.

All the steps followed are explained in the code.